Foreword by Ed Burns

Sample Chapter

JavaServer Faces IN ACTION

Kito Mann

MANNING

JavaServer Faces in Action

by Kito Mann

Sample Chapter 1

brief contents

ONLINE EXTENSION

The five chapters in part 5 (plus four additional appendixes) are not included in the print edition. They are available for download in PDF format from the book's web page to owners of this book. For free access to the online extension please go to www.manning.com/mann.

1 Introducing JavaServer Faces

This chapter covers

- What JavaServer Faces is, and what it's not
- Foundation technologies (HTTP, servlets, portlets, JavaBeans, and JSP)
- How JavaServer Faces relates to existing web development frameworks
- Building a simple application

Welcome to *JavaServer Faces in Action*. JavaServer Faces (JSF, or simply "Faces") makes it easy to develop web applications by bringing support for rich, powerful user interface components (such as text boxes, list boxes, tabbed panes, and data grids) to the web development world. A child of the Java Community Process,[1] JSF is destined to become a part of Java 2 Enterprise Edition (J2EE). This book will help you understand exactly what JSF is, how it works, and how you can use it in your projects today.

1.1 It's a RAD-ical world

A popular term in the pre-Web days was *Rapid Application Development* (RAD). The main goal of RAD was to enable you to build powerful applications with a set of reusable components. If you've ever used tools like Visual Basic, PowerBuilder, or Delphi, you know that they were a major leap forward in application development productivity. For the first time, it was easy to develop complex user interfaces (UIs) and integrate them with data sources.

You could drag application widgets—UI controls and other components—from a palette and drop them into your application. Each of these components had properties that affected their behavior. (For example, `font` is a common property for any control that displays text; a data grid might have a `dataSource` property to represent a data store.) These components generated a set of events, and event handlers defined the interaction between the UI and the rest of the application. You had access to all of this good stuff directly from within the integrated development environment (IDE), and you could easily switch between design and code-centric views of the world.

RAD tools were great for developing full-fledged applications, but they were also quite useful for rapid prototyping because they could quickly create a UI with little or no code. In addition, the low barrier to entry allowed both experienced programmers and newbies to get immediate results.

These tools typically had four layers:

- An underlying component architecture
- A set of standard widgets
- An application infrastructure
- The tool itself

[1] The *Java Community Process* (JCP) is the public process used to extend Java with new application programming interfaces (APIs) and other platform enhancements. New proposals are called *Java Specification Requests* (JSRs).

The underlying component architectures were extensible enough to spawn an industry of third-party component developers like Infragistics and Developer Express.

Of course, the RAD philosophy never went away—it just got replaced by other hip buzzwords. It's alive and well in some Java IDEs and other development environments like Borland Delphi and C++Builder. Those environments, however, stop short of using RAD concepts for web projects. The adoption of RAD in the web development world has been remarkably slow.

This sluggishness is due in part to the complexity of creating such a simple, cohesive view of application development in a world that isn't simple or cohesive. Web applications are complex if you compare them to standard desktop applications. You've got a ton of different resources to manage—pages, configuration files, graphics, and code. Your users may be using different types of browsers running on different operating systems. And you have to deal with HTTP, a protocol that is ill suited for building complex applications.

The software industry has become good at masking complexity, so it's no surprise that many RAD web solutions have popped up over the last few years. These solutions bring the power of visual, component-oriented development to the complex world of web development. The granddaddy is Apple's WebObjects,[2] and Microsoft has brought the concept to the mainstream with Visual Studio.NET and ASP.NET Web Forms. In the Java world, many frameworks have emerged, several of them open source. Some have tool support, and some don't.

However, the lack of a *standard* Java RAD web framework is a missing piece of the Java solution puzzle—one that Microsoft's. NET Framework has covered from day one. JavaServer Faces was developed specifically to fill in that hole.

1.1.1 So, what is JavaServer Faces?

In terms of the four layers of a RAD tool, JavaServer Faces defines three of them: a component architecture, a standard set of UI widgets, and an application infrastructure. JSF's component architecture defines a common way to build UI widgets. This architecture enables standard JSF UI widgets (buttons, hyperlinks, checkboxes, text fields, and so on), but also sets the stage for third-party components. Components are event oriented, so JSF allows you to process client-generated events (for instance, changing the value of a text box or clicking on a button).

Because web-based applications, unlike their desktop cousins, must often appease multiple clients (such as desktop browsers, cell phones, and PDAs), JSF

[2] WebObjects has a full-fledged environment that includes a J2EE server, web services support, and object persistence, among other things.

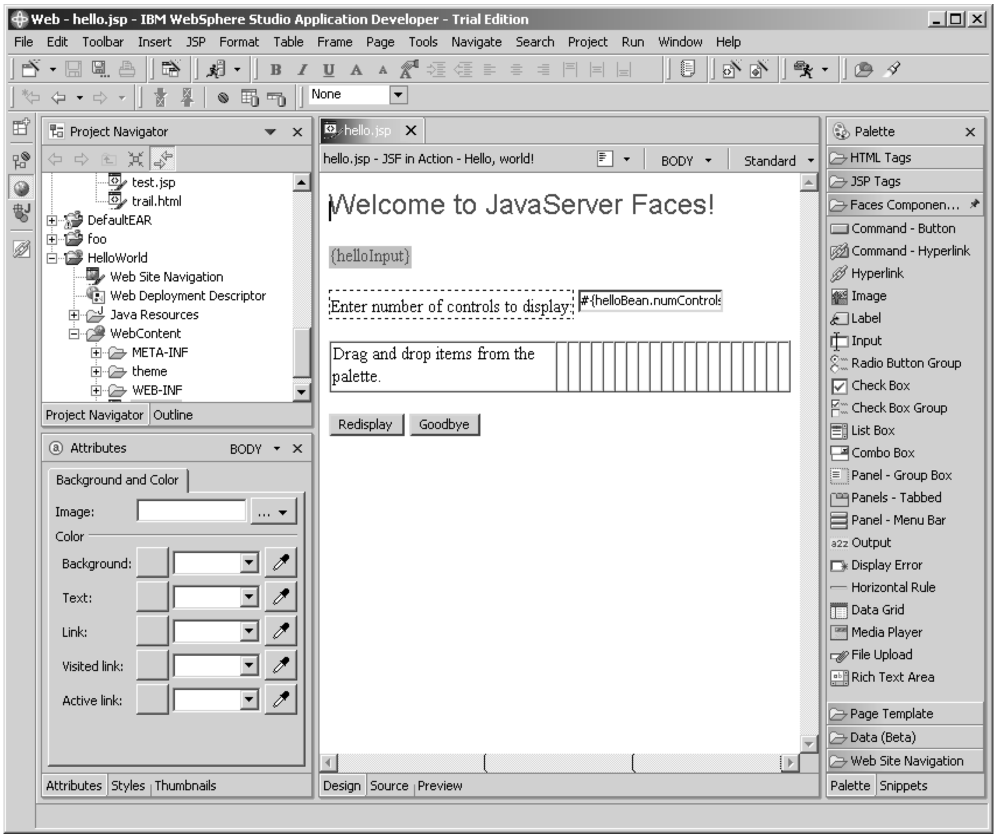
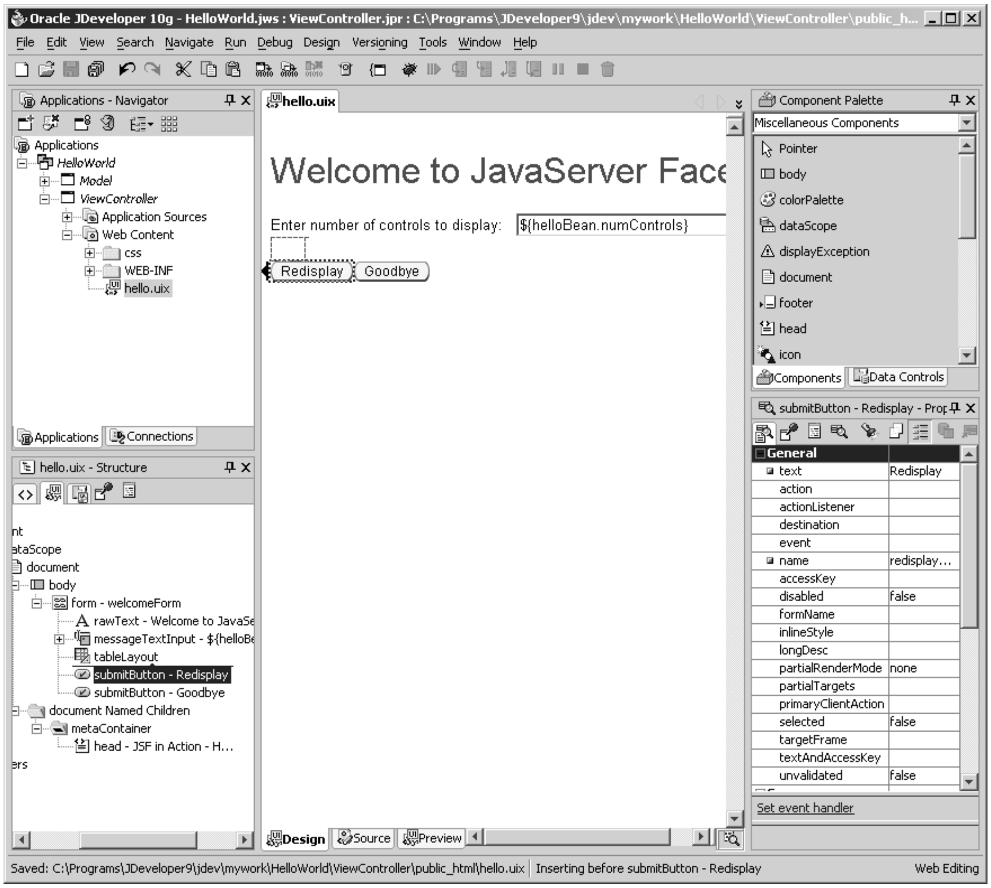


Figure 1.1 IBM's WebSphere Application Developer (WSAD) has been expanded to support JSF applications in addition to the seemingly endless amount of other technologies it supports. You can visually build JSF applications, and mix-and-match other JSP tag libraries using WSAD's familiar Eclipse-based environment.

has a powerful architecture for displaying components in different ways. It also has extensible facilities for validating input (the length of a field, for example) and converting objects to and from strings for display.

Faces can also automatically keep your UI components in synch with Java objects that collect user input values and respond to events, which are called *backing beans*. In addition, it has a powerful navigation system and full support for multiple languages. These features make up JSF's application infrastructure—basic building blocks necessary for any new system.

JavaServer Faces defines the underpinnings for tool support, but the implementation of specific tools is left to vendors, as is the custom with Java. You have



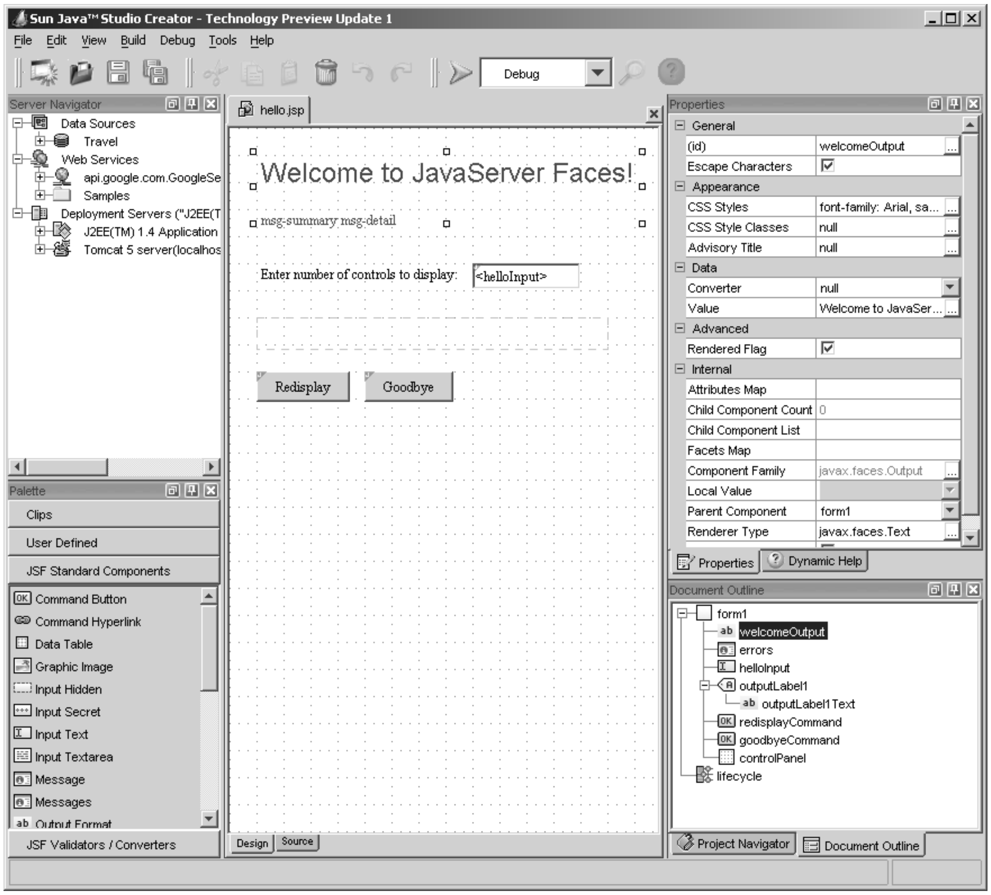
General	
text	Redisplay
action	
actionListener	
destination	
event	
name	redisplay...
accessKey	
disabled	false
formName	
inlineStyle	
longDesc	
partialRenderMode	none
partialTargets	
primaryClientAction	
selected	false
targetFrame	
textAndAccessKey	
unvalidated	false

Saved: C:\Programs\JDeveloper9\jdev\mywork\HelloWorld\ViewController\public_html\hello.uix | Inserting before submitButton - Redisplay

Figure 1.2 Oracle's JDeveloper [Oracle, JDeveloper] will have full-fledged support for JSF, complete with an extensive array of UIX components, which will integrate with standard JSF applications. It will also support using JSF components with its Application Development Framework (ADF) [Oracle, ADF]. (This screen shot was taken with UIX components available with JDeveloper 10g, which are the basis of JSF support in the next version of JDeveloper.)

a choice of tools from industry leaders that allow you to visually lay out a web UI in a way that's quite familiar to users of RAD development tools such as Visual Studio. NET. (Figures 1.1, 1.2, and 1.3 show what Faces development looks like in IDEs from IBM, Oracle, and Sun, respectively.) Or, if you prefer, you can develop Faces applications without design tools.

Just in case all of this sounds like magic, we should point out a key difference between JavaServer Faces and desktop UI frameworks like Swing or the Standard Widget Toolkit (SWT): JSF runs on the *server.* As such, a Faces application will run



General	
(id)	welcomeOutput
Escape Characters	✓
Appearance	
CSS Styles	font-family: Arial, sa...
CSS Style Classes	null
Advisory Title	null
Data	
Converter	null
Value	Welcome to JavaSer...
Advanced	
Rendered Flag	✓
Internal	
Attributes Map	
Child Component Count	0
Child Component List	
Facets Map	
Component Family	javax.faces.Output
Local Value	
Parent Component	form1
Renderer Type	javax.faces.Text

Figure 1.3 Sun's Java Studio Creator [Sun, Creator] is an easy-to-use, visually based environment for building JavaServer Faces applications. You can easily switch between designing JSF pages visually, editing the JSP source, and writing associated Java code in an environment that should seem familiar to users of Visual Studio.NET, Visual Basic, or Delphi.

Widget Toolkit (SWT): JSF runs on the *server.* As such, a Faces application will run in a standard Java web container like Apache Tomcat [ASF, Tomcat], Oracle Application Server [Oracle, AS], or IBM WebSphere Application Server [IBM, WAS], and display HTML or some other markup to the client.

If you click a button in a Swing application, it will fire an event that you can handle directly in the code that resides on the desktop. In contrast, web browsers don't know anything about JSF components or events; they just know how to

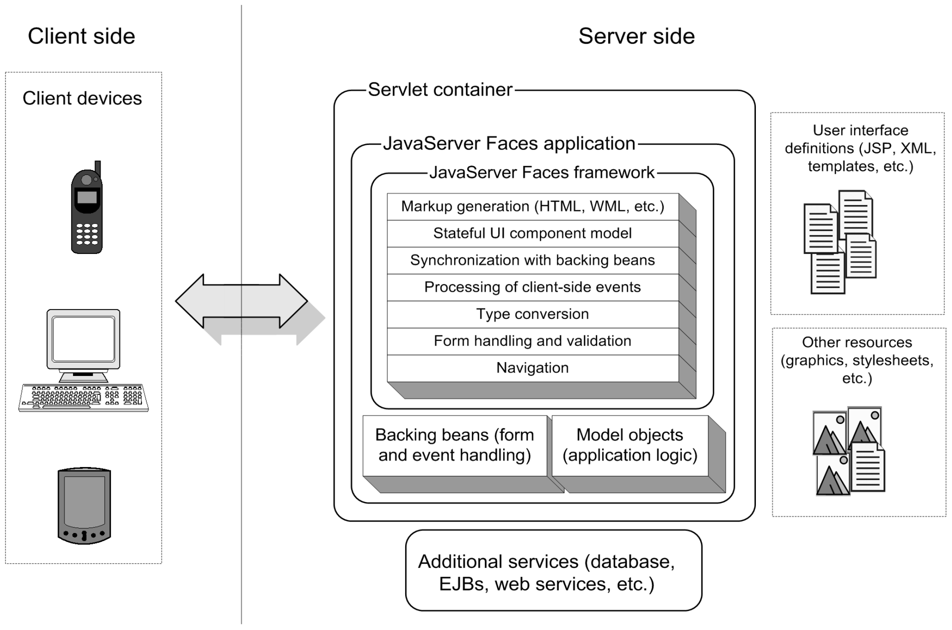

Client side
Server side
Client devices
Servlet container
JavaServer Faces application
JavaServer Faces framework
Markup generation (HTML, WML, etc.)
Stateful UI component model
Synchronization with backing beans
Processing of client-side events
Type conversion
Form handling and validation
Navigation
Backing beans (form and event handling)
Model objects (application logic)
Additional services (database, EJBs, web services, etc.)
User interface definitions (JSP, XML, templates, etc.)
Other resources (graphics, stylesheets, etc.)

Figure 1.4 A high-level view of a JavaServer Faces application. JSF makes web development easy by providing support for UI components and handling a lot of common web development tasks.

display HTML.[3] So when you click on a button in a Faces application, it causes a request to be sent from your web browser to the server. Faces is responsible for translating that request into an event that can be processed by your application logic on the server. It's also responsible for making sure that every UI widget you've defined on the server is properly displayed to the browser.

Figure 1.4 shows a high-level view of a Faces application. You can see that the application runs on the server and can integrate with other subsystems, such as Enterprise JavaBeans (EJB) services or databases. However, JSF provides many additional services that can help you build powerful web applications with less effort.

JavaServer Faces has a specific goal: to make web development faster and easier. It allows developers to think in terms of components, events, backing beans, and their interactions, instead of requests, responses, and markup. In other words, it masks a lot of the complexities of web development so that developers can focus on what they do best—build applications.

[3] Technically, they do a lot of other things, like execute JavaScript or VBScript, display XML and XHTML, and so on.

NOTE JSF is a technology, and this book covers it as thoroughly as possible in several hundred pages. Tools sugarcoat a lot of pieces of the development puzzle with graphical user interface (GUI) designers that generate JSP, screens that edit configuration files, and so forth. Throughout this book, we'll show you the real Java, JSP, and XML code and configuration that JSF uses, while giving you a sense of where the tools can make your life easier. With this approach, you'll have a full understanding of what an IDE does behind the scenes, which is extremely useful for maintenance of your application after it's built, and also for situations where you need to move from one IDE vendor to another. And, of course, if you don't like IDEs at all, knowing how things actually work is essential. (If you are a big IDE fan, don't worry—we show screen shots of different tools throughout the book, and online extension appendix B covers three of them in detail).

1.1.2 Industry support

One of the best things about the Java Community Process (JCP), Sun Microsystems's way of extending Java, is that a lot of great companies, organizations, and individuals are involved. Producing a spec through the JCP isn't exactly speedy, but the work can be quite good. JavaServer Faces was introduced as Java Specification Request (JSR) 127 by Sun in May 2001; the final version of the specification, JSF 1.0, was released on March 3, 2004, and JSF 1.1 (a maintenance release) arrived on May 27th, 2004. The companies and organizations (other than Sun) involved in developing Faces include the Apache Software Foundation, BEA Systems, Borland Software, IBM, Oracle, Macromedia, and many others.

The products developed by these companies can be put into three categories (many fit in more than one): J2EE containers, development tools, and UI frameworks. Because JavaServer Faces is a UI component framework that works with tools and runs inside J2EE containers, this makes good sense. What's significant is the fact that the group includes many industry heavyweights. This means that you can expect JSF to have a lot of industry support. And if your vendor doesn't support JSF, you can download Sun's reference implementation for free [Sun, JSF RI].

To keep up with the latest JSF news, articles, products and vendors, check out JSF Central [JSF Central], a community site run by the author.

1.2 The technology under the hood

All JSF applications are standard Java web applications. Java web applications speak the Hypertext Transfer Protocol (HTTP) via the Servlet API and typically use some sort of display technology, such as JavaServer Pages (JSP), as shown in figure 1.4. The display technology is used to define UIs that are composed of components that interact with Java code. Faces applications can also work inside of portlets, which are similar to servlets. JSF's component architecture uses JavaBeans for exposing properties and event handling.

In this section, we briefly describe these technologies and explain how they relate to JSF. If you're already familiar with Java web development basics and understand how they relate to JSF, you may want to skip this section.

1.2.1 Hypertext Transfer Protocol (HTTP)

Diplomats and heads of state come from many different cultures and speak many different languages. In order to communicate, they follow specific rules of ceremony and etiquette, called *protocols*. Following protocols helps to ensure that they can correspond effectively, even though they come from completely different backgrounds.

Computers use protocols to communicate as well. Following an established set of rules allows programs to communicate regardless of the specific software, hardware, or operating system.

The World Wide Web (WWW) started as a mechanism for sharing documents. These documents were represented via the Hypertext Markup Language (HTML) and allowed people viewing the documents to easily move between them by simply clicking on a link. To serve up documents and support this hyperlinking capability, the Hypertext Transfer Protocol (HTTP) was developed. It allowed any web browser to grab documents from a server in a standard way.

DEFINITION The Web was originally designed for *static* content such as academic documents, which do not change often. In contrast, *dynamic* content, such as stock information or product orders, changes often. Dynamic content is what applications usually generate.

HTTP is a simple protocol—it's based on text headers. A client sends a request to a server, and the server sends a response back to the browser with the requested document attached. The server is dumb[4]—it doesn't remember anything about the client if another document is requested. This lack of memory means that HTTP is a "stateless" protocol; it maintains no information about the client between requests.

[4] Web servers have grown to be quite sophisticated beasts, but initially they were pretty simple.

The stateless nature of HTTP means that it's able to scale well (it is, after all, *the* protocol of the Internet, and the Internet is a huge place). This property isn't a problem for the static documents that HTTP was originally developed to serve.

But imagine what it'd be like if a valet parked your car but didn't give you a ticket and didn't remember your face. When you came back, he'd have a hard time figuring out which car to retrieve. That's what it's like to develop an application in a stateless environment. To combat this problem, there are two possibilities: cookies and URL rewriting. They're both roughly the same as the valet giving you a ticket and keeping one himself.

No matter what language you use, if you're writing a web application, it will use HTTP. Servlets and JSP were developed to make it easier to build applications on top of the protocol. JavaServer Faces was introduced so that developers can forget that they're using the protocol at all.

1.2.2 Servlets

HTTP is great for serving up static content, and web servers excel at that function out of the box. But creating dynamic content requires writing code. Even though HTTP is simple, it still takes some work to write programs that work with it. You have to parse the headers, understand what they mean, and then create new headers in the proper format. That's what the Java Servlet application programming interface (API) is all about: providing an object-oriented view of the world that makes it easier to develop web applications.[5] HTTP requests and responses are encapsulated as objects, and you get access to input and output streams so that you can read a user's response and write dynamic content. Requests are handled by *servlets*—objects that handle a particular set of HTTP requests.

A standard J2EE web application is, by definition, based on the Servlet API. Servlets run inside a *container*, which is essentially a Java application that performs all of the grunt work associated with running multiple servlets, associating the resources grouped together as a web application, and managing all sorts of other services. The most popular servlet container is Tomcat [ASF, Tomcat], but J2EE application servers such as IBM WebSphere [IBM, WAS] and the Sun Java System Application Server [Sun, JSAS] provide servlet containers as well.

As we mentioned in the previous section, one of the big problems with HTTP is that it's stateless. Web applications get around this problem through the use of

[5] Technically, the Servlet API can be used to provide server functionality in any request/response environment—it doesn't necessarily have to be used with HTTP. In this section, we're referring to the `java.servlet.http` package, which was designed specifically for processing HTTP requests.

sessions—they make it seem as if the users are always there, even if they're not. Sessions are one of the biggest benefits that the Servlet API provides. Even though behind the scenes they make use of cookies or URL rewriting, the programmer is shielded from those complexities.

The Servlet API also provides lots of other goodies, like security, logging, life-cycle events, filters, packaging and deployment, and so on. These features all form the base of JavaServer Faces. As a matter of fact, JSF is implemented as a servlet, and all JSF applications are standard J2EE web applications.

JSF takes things a bit further than the Servlet API, though. Servlets cover the basic infrastructure necessary for building web applications. But at the end of the day, you still have to deal with requests and responses, which are properties of the underlying protocol, HTTP. JSF applications have UI components, which are associated with backing beans and can generate events that are consumed by application logic. Faces uses the Servlet API for all of its plumbing, but the developer gets the benefit of working at a higher level of abstraction: You can develop web applications without worrying about HTTP or the specifics of the Servlet API itself.

1.2.3 Portlets

Most web applications serve dynamic content from a data store—usually a database. (Even if the business logic is running on another type of server, like an EJB or Common Object Request Broker Architecture [CORBA] server, eventually some code talks to a database.) Since the early days of the Web, however, there has been a need for software that aggregates information from different data sources into an easy-to-use interface. These types of applications, called *portals*, were originally the domain of companies like Netscape and Yahoo! However, more and more companies now realize that the same concept works well for aggregating information from different internal data sources for employee use.

So a variety of vendors, including heavyweights like IBM, BEA, and Oracle, offer portal products to simplify this task. Each data source is normally displayed in a region within a web page that behaves similarly to a window—you can close the region, customize its behavior, or interact with it independent of the rest of the page. Each one of these regions is called a *portlet.*

Each of these vendors developed a completely different API for writing portlets that work with their portal products. In order to make it easier to develop portlets that work in multiple portals, the JCP developed the Portlet specification [Sun, Portlet], which was released in late 2003. All of the major portal vendors (including Sun, BEA, IBM, and Oracle) and open source organizations like the Apache Software Foundation have announced support for this specification in their portal products.

The Portlet specification defines the Portlet API, which, like the Servlet API, defines a lot of low-level details but doesn't simplify UI development or mask HTTP. That's where JSF comes into the picture; it was developed so that it can work with the Portlet API (which is similar in many ways to the Servlet API). You can use ordinary JSF components, event handling, and other features inside portlets, just as you can inside servlets.

NOTE Throughout this book, we mostly talk about JSF in relation to servlets. However, most of our discussions apply to portlets as well.

1.2.4 JavaBeans

Quite a few Java web developers think that JavaBeans are simply classes with some properties exposed via getter and setter methods (accessors and mutators). For example, a Java class with the methods `getName` and `setName` exposes a read-write property called `name`. However, properties are just the tip of the iceberg; JavaBeans is a full-fledged component architecture designed with tool support in mind.

This is significant, because it means there's a lot more to it than just properties. JavaBeans conform to a set of patterns that allow other Java classes to dynamically discover events and other metadata in addition to properties. As a matter of fact, JavaBeans is the technology that enables Swing and makes it possible for IDEs to provide GUI builders for desktop applications and applets. Using JavaBeans, you can develop a component that not only cooperates nicely with a visual GUI builder but also provides a specialized wizard (or *customizer*) to walk the user through the configuration process. JavaBeans also includes a powerful event model (the same one used with Swing and JSF components), persistence services, and other neat features.

Understanding the power of JavaBeans will help you comprehend the full power of JSF. Like Swing components, every JSF component is a full-fledged JavaBean. In addition, Faces components are designed to work with backing beans—objects that are implemented as JavaBeans and also handle events.

If you're just planning to write application code or build UIs, then a basic knowledge of JavaBeans (mutators and accessors) is sufficient. If you're going to be developing custom components, a deep understanding of JavaBeans will make your life much easier.

1.2.5 JSP and other display technologies

Servlets are great low-level building blocks for web development, but they don't adequately simplify the task of displaying dynamic content. You have to manually write out the response to every request.

Let's say that every line of HTML you were sending was written as a separate line of Java code. You have about 30 pages in your application, and each page has about 80 lines of HTML. All of the sudden, you have 2400 lines of code that looks a lot like this:

```
out.println("This is a <b>really</b> repetitive task, and \"escaping\"" +
         " text is a pain. ");
```

This is really tedious work, especially because you have to escape a lot of characters, and it's hard to quickly make changes. Clearly there has to be a better way.

To solve this problem, Sun introduced JavaServer Pages (JSP) as a standard template mechanism. JavaServer Pages look like an HTML page, but they have special tags that do custom processing or display JavaBean values, and can also have Java code embedded in them.[6] Ultimately, they behave like a servlet that looks a lot like the previous code snippet. The JSP translator does the boring work so that you don't have to.

You can create your own custom tags[7] to perform additional processing (such as accessing a database), and there's a useful set of standard tags called the JavaServer Pages Standard Tag Library (JSTL) [Sun, JSTL]. The idea is that you can define the UI with HTML-like tags, not Java code.

Even though JSP is the industry standard display technology, you can choose among many alternatives. You could use a full Extensible Markup Language/ Extensible Style Sheet Language Transformations (XML/XSLT) approach with something like Cocoon [ASF, Cocoon], or a stricter template-based approach like Velocity [ASF, Velocity] or WebMacro [WebMacro]. Many other options are available as well.

One of the key design goals of JSF was to avoid relying on a particular display technology. So JSF provides pluggable interfaces that allow developers to integrate

[6] The ability to have Java code embedded in JSPs is considered bad design (and a bad practice) by many and is the topic of one of those huge religious wars. The main argument is that it doesn't enforce separation between display and business logic. That "feature" is one of the main reasons there are different choices for display technologies. In JSP 2.0, you can turn off this feature.

[7] Custom tags are technically called "custom actions," but we use the more common term "custom tags" throughout this book.

it with various display technologies. However, because JSF is a standard Java technology, and so is JSP, it's no surprise that Faces comes with a JSP implementation (via custom tags) right out of the box. And because JSP is the only display technology that *must* be integrated with JavaServer Faces, most of the examples in this book use JSP as well.

1.3 Frameworks, frameworks, frameworks

Earlier, we said that JavaServer Faces is a "framework" for developing web-based UIs in Java. Frameworks are extremely common these days, and for a good reason: they help make web development easier. Like most Java web frameworks, JSF enforces a clean separation of presentation and business logic. However, it focuses more on the UI side of things and can be integrated with other frameworks, like Struts.

1.3.1 Why do we need frameworks?

As people build more and more web applications, it becomes increasingly obvious that although servlets and JSPs are extremely useful, they can't handle many common tasks without tedious coding. Frameworks help simplify these tasks.

The most basic of these tasks is form processing. HTML pages have forms, which are collection of user input controls like text boxes, lookup lists, and checkboxes. When a user submits a form, all of the data from the input fields is sent to the server. A text field in HTML might look like this:

```
<input maxLength=256 size=55 name="userName" value="">
```

In a standard servlet application, the developer must retrieve those values directly from the HTTP request like this:

```
String userName = (String)request.getParameter("userName");
```

This can be tedious for large forms, and because you're dealing directly with the value sent from the browser, the Java code must also make sure all of the request parameters are valid. In addition, each one of these parameters must be manually associated with the application's objects.

Forms are just one example of tasks that servlets and JSP don't completely solve. Web applications have to manage a lot of pages and images, and referencing all of those elements within larger applications can become a nightmare if you don't have a central way of managing it.

Management of the page structure is another issue. Although JSP provides a simple mechanism for creating a dynamic page, it doesn't provide extensive support for *composing* a page out of smaller, reusable parts. Other fun things that servlets don't handle include internationalization, type conversion, and error handling.

To handle all of these tasks in a simplified manner, several frameworks have emerged. Some of the more popular ones are Struts [ASF, Struts] and WebWork [OpenSymphony, WebWork]. The goal of any framework is to facilitate development by handling many common tasks.

1.3.2 She's a Model 2

Basic organization and management services are a necessity for larger web applications, but they need structure as well. Most web frameworks, including JSF, enforce some variation of the Model-View-Controller (MVC) design pattern. To understand exactly what MVC is, let's look at a driving analogy.

When you're driving down the highway in one direction, there's usually a median between you and the traffic heading in the opposite direction. The median is there for a good reason—fast traffic moving in opposite directions doesn't mix too well. Without the median, a rash of accidents would inevitably result.

Applications have similar issues: Code for business logic doesn't mix too well with UI code. When the two are mixed, applications are much harder to maintain, less scalable, and generally more brittle. Moreover, you can't have one team working on presentation code while another works on business logic.

The MVC pattern is the standard solution to this problem. When you watch a story on the news, you view a version of reality. An empirical event exists, and the news channel is responsible for interpreting the event and broadcasting that interpretation. Even though you see the program on your TV, a distinct difference lies between what actually took place, how people doing the reporting understand it, and what you're seeing on your TV. The news channel is *controlling* the interaction between the TV program—the *view*—and the actual event—the model. Even though you may be watching the news on TV, the same channel might be broadcasting via the Internet or producing print publications. These are alternate views. If the pieces of the production weren't separate, this wouldn't be possible.

In software, the view is the presentation layer, which is responsible for interacting with the user. The model is the business logic and data, and the controller is the application code that responds to user events and integrates the model and view. This architecture ensures that the application is loosely coupled, which reduces dependencies between different layers.

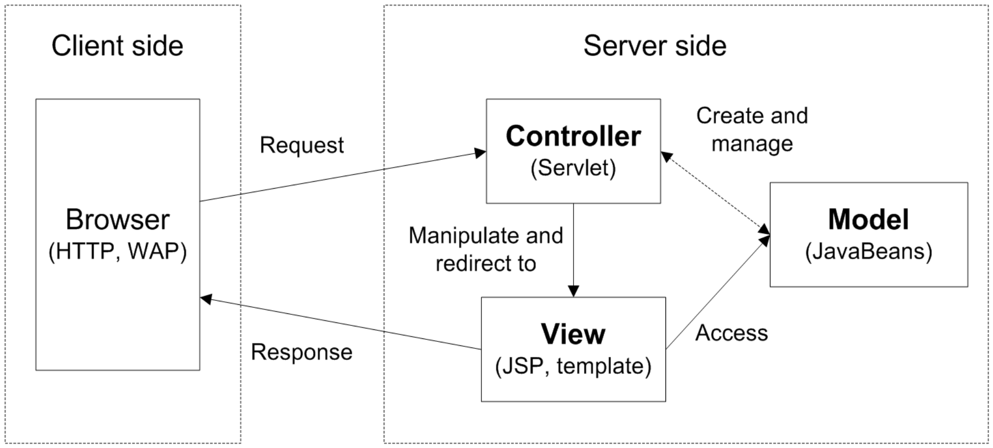


Figure 1.5 Most web frameworks use some variation of the Model 2 design pattern.

Model 2 (shown in figure 1.5) is a variation of MVC that's specific to web applications. The basic point is that:

- The model can consist of plain old Java objects (POJOs), EJBs, or something else.
- The view can be JSPs or some other display technology.
- The controller is always implemented as a servlet.

So if the JSP page contains an error, it doesn't affect the application code or the model. If there's an error in the model, it doesn't affect the application code or the JSP page. This separation allows for unit testing at each layer, and also lets different parties work with the layers independently. For instance, a front-end developer can build a JSP before the business objects and the application code are complete. Portions of some layers can even be integrated before all three have been completed.

These benefits are exactly why most frameworks, including JSF, support some variation of the MVC design pattern.

1.3.3 JSF, Struts, and other frameworks

Let's face it: there are a lot of Java web frameworks available. Some of them, like Struts [ASF, Struts] and WebWork [OpenSymphony, WebWork], help with form processing and other issues such as enforcing Model 2, integrating with data sources, and controlling references to all of the application's resources centrally via

XML configuration files. These *foundation frameworks* provide extensive underpinnings but don't mask the fundamental request/response nature of HTTP.

Other frameworks, like Tapestry [ASF, Tapestry], Oracle's Application Development Framework (ADF) UIX [Oracle, ADF UIX], and SOFIA [Salmon, SOFIA], provide a UI component model and some sort of event processing. The purpose of these *UI frameworks*, which include JSF, is to simplify the entire programming model. Often, foundation and UI frameworks have overlapping functionality.

To understand this overlap, you can think of web application infrastructure as a stack of services. The services close to the bottom of the stack don't abstract too many details of the underlying protocol; they're more like plumbing. The services toward the top of the stack hide more of the gory details; they provide higher levels of abstraction. The lowest services are handled by web servers, the Servlet API, and JSP. Most frameworks provide some subsection of the additional services. Figure 1.6 shows this stack in relation to JSF, Struts, servlets, JSP, and a traditional web server.

You can see from the figure that JSF supports enough services to make it quite powerful by itself, and in many cases, it's all you'll need. Subsequent releases of Faces will most likely cover additional services as well.

However, even though Faces overlaps with frameworks like Struts, it doesn't necessarily replace them. (As a matter of fact, the lead developer of Struts, Craig McClanahan, was instrumental in the development of JavaServer Faces.) If you integrate the two, you get access to all the services of the stack (chapter 14 covers Struts integration). You can also use JSF with other frameworks like Spring [Spring-Faces].

For UI-oriented frameworks, JSF may overlap with a large set of their functionality. Some of those projects have pledged support for JSF in future versions. Faces has the distinction of being developed by a consortium of industry heavyweights through the JCP and will be part of J2EE. As a result, it enjoys heavy tool support and will ship standard with many J2EE servers.

1.4 Components everywhere

Sadly, overuse of the term "component" is rampant in the industry today. An operating system is a component, an application is a component, EJBs are components, a library is a component, and so is the kitchen sink. Numerous books about components are available, and the good ones point out that many definitions exist.

The excessive use of this word isn't that strange if you know what it really means. If you look up "component" in the dictionary, you'll see that it's a

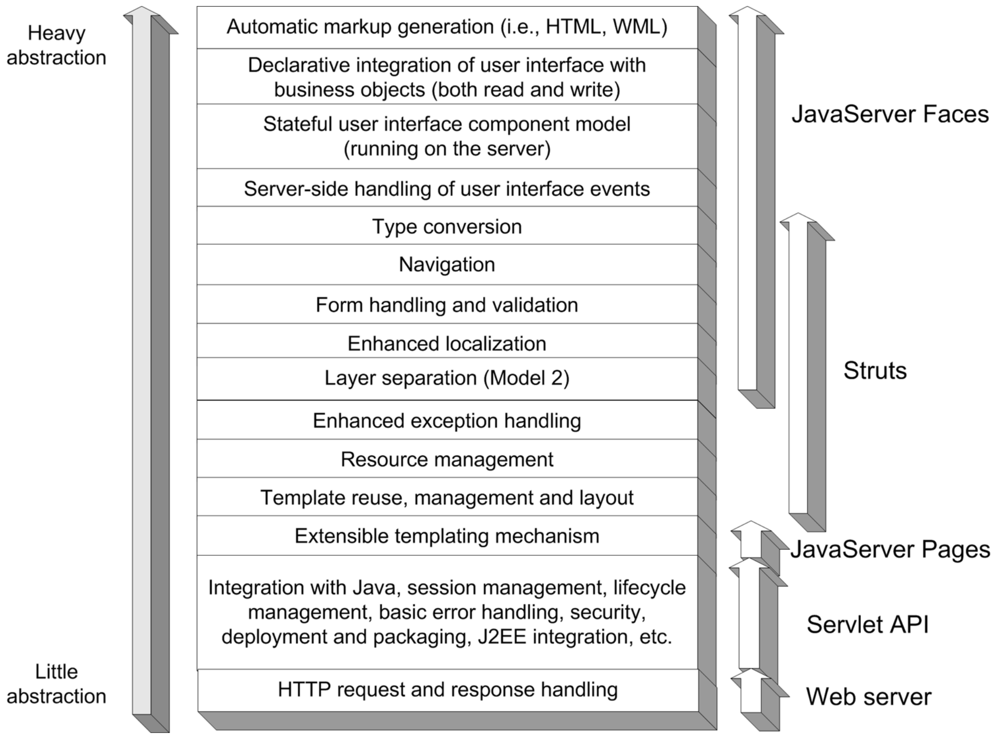

Heavy abstraction
Little abstraction
Automatic markup generation (i.e., HTML, WML)
Declarative integration of user interface with business objects (both read and write)
Stateful user interface component model (running on the server)
Server-side handling of user interface events
Type conversion
Navigation
Form handling and validation
Enhanced localization
Layer separation (Model 2)
Enhanced exception handling
Resource management
Template reuse, management and layout
Extensible templating mechanism
Integration with Java, session management, lifecycle management, basic error handling, security, deployment and packaging, J2EE integration, etc.
HTTP request and response handling
JavaServer Faces
Struts
JavaServer Pages
Servlet API
Web server

Figure 1.6 Web application infrastructure can be viewed as a stack of services. The services on the bottom provide basic plumbing but little abstraction. The services at the top of the stack provide more abstraction. Having all of the services together is extremely powerful.

synonym for *constituent*—a part of a whole. So, if you use the literal meaning of the word, an operating system really is a component in the context of a distributed application.

What's funny is that conceptually, a kitchen sink has more in common with Faces components than an operating system does. If you remodel your kitchen, you get to pick out a kitchen sink. You don't have to *build* it from scratch—you just have to pick a sink that fulfills your requirements: size, color, material, number of bowls, and so on. The same thing goes for other kitchen items, like cabinets and countertops. All of these components have specific interfaces that allow them to integrate with one another, but they depend on specific environmental services (plumbing, for instance). The end result may be unique, but the whole is made up of independent, reusable parts.

If we take the concepts of kitchen components and apply them to software, we end up with this definition:

DEFINITION A *software component* is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties [Szyperski].

The "context dependencies" in a kitchen are things like the room itself, plumbing, and electrical circuits. In essence, the context is the container for all of the components. A *container* is a system that hosts components and provides a set of services that allow those components to be manipulated. Sometimes that manipulation is within an IDE (during *design time*); sometimes it's in a deployment environment, like a J2EE server (during *runtime*).

The phrase "deployed independently" means that a component is a self-contained unit and can be installed into a container. Kitchen sinks are individual, self-contained items made to fit into a countertop.

When you remodel your kitchen, you hire a contractor, who assembles the components you've selected (cabinets, drawers, sink, and so on) into a full-fledged kitchen. When we build software using component architectures, we assemble various components to create a working software system.

JSF components, Swing components, servlets, EJBs, JavaBeans, ActiveX controls, and Delphi Visual Component Library (VCL) components all fit this definition. But these components concentrate on different things. JSF and Swing components are aimed solely at UI development, while ActiveX and VCL controls may or may not affect the UI. Servlets and EJBs are much more *coarse-grained*—they provide a lot of functionality that's more in the realm of application logic and business logic.

Because JSF is focused on UI components, let's narrow our component definition appropriately:

DEFINITION A *UI component*, or *control*, is a component that provides specific functionality for interacting with an end user. Classic examples include toolbars, buttons, panels, and calendars.

If you've done traditional GUI development, then the concept of a UI component should be quite familiar to you. What's great about JavaServer Faces is that it brings a standard UI component model to the web world. It sets the stage for things desktop developers take for granted: a wide selection of packaged UI functionality with extensive tools support. It also opens the door for creating custom components that handle tasks specific to a particular business domain—like a report viewer or an interest calculator.

1.5 Hello, world!

Now that you have a basic understanding of the problems JavaServer Faces is meant to solve, let's begin with a simple Faces application. This section assumes you're familiar with Java web applications and JSP. (For more about these technologies, see section 1.2 for an overview.) We'll dissect a simple HTML-based web application that has two pages: hello.jsp and goodbye.jsp.

The hello.jsp page does the following:

- Displays the text "Welcome to JavaServer Faces!"
- Has a single form with a text box that requires an integer between 1 and 500
- Stores the last text box value submitted in a JavaBean property called `numControls`
- Has a grid underneath the text box
- Has a button labeled "Redisplay" that when clicked adds `numControls` output UI components to the grid (clearing it of any previous UI components first)
- Has a button labeled "Goodbye" that displays goodbye.jsp if clicked

The goodbye.jsp page does the following:

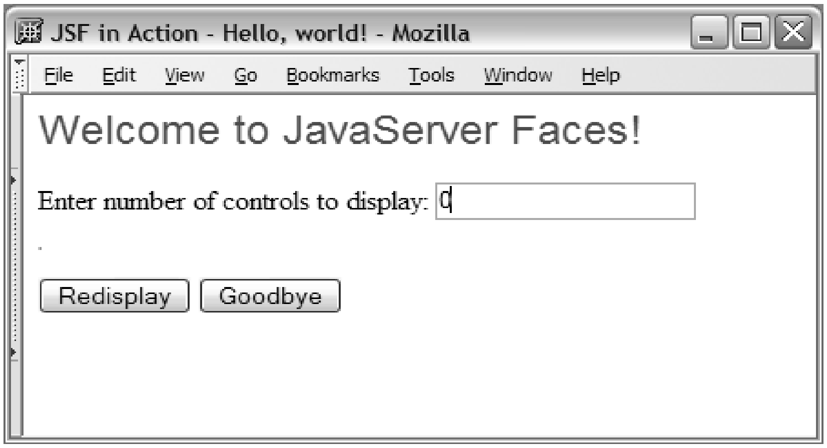
- Displays the text "Goodbye!"
- Displays the value of the JavaBean property `numControls`

JSF performs most of the work of our Hello, world! application, but in addition to the JSP pages, there are a few other requirements:

- The `HelloBean` backing bean class
- A Faces configuration file
- A properly configured deployment descriptor

Some tools will simplify creation of some or all of these requirements, but in this section, we'll examine the raw files in detail.

Before we get into those details, let's see what Hello, world! looks like in a web browser. The application starts with hello.jsp, as shown in figure 1.7. The text box on this page is associated with a JavaBean property of the `HelloBean` class;


JSF in Action - Hello, world! - Mozilla
File Edit View Go Bookmarks Tools Window Help
Redisplay
Goodbye

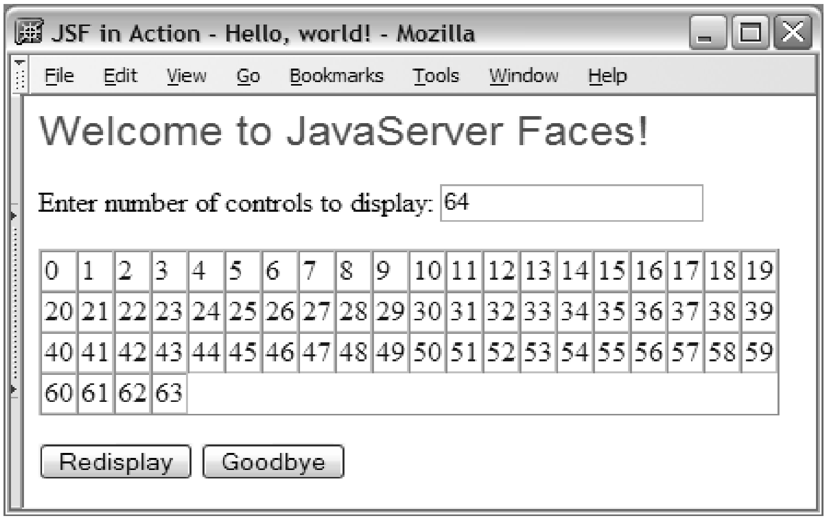
Welcome to JavaServer Faces!

Enter number of controls to display: 0

Figure 1.7 The Hello, world! application before any data has been submitted.

when someone enters a value into this box, the property will be updated automatically (if the value is valid).

If you enter the number “64” into the text box and click the Redisplay button, the page redisplays as shown in figure 1.8—a total of 64 UI components are displayed in the grid. If you clear the text box and click the Redisplay button, you’ll get a validation error, as shown in figure 1.9. You’ll also get a validation error if

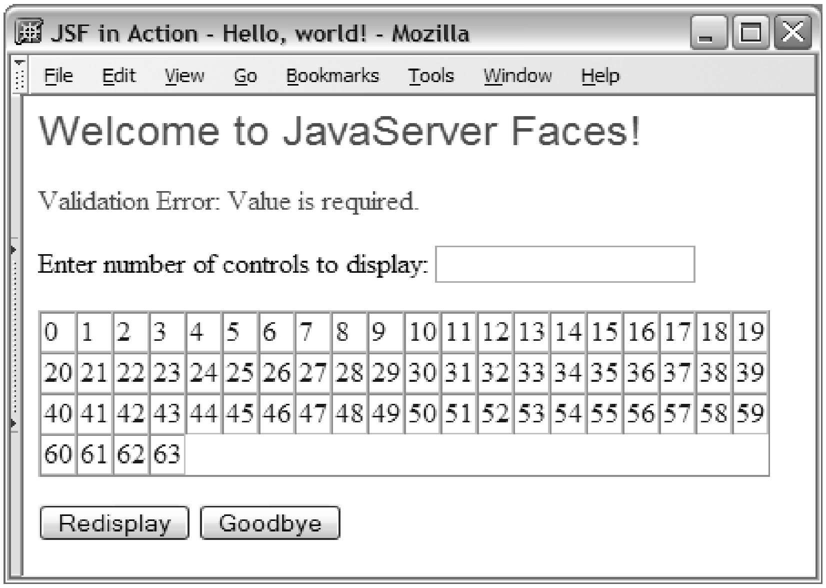

JSF in Action - Hello, world! - Mozilla
File Edit View Go Bookmarks Tools Window Help
Redisplay
Goodbye

Welcome to JavaServer Faces!

Enter number of controls to display: 64

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
60	61	62	63																

Figure 1.8 The Hello, world! application after you enter "64" and click the Redisplay button. The grid is populated with 64 UI components.


JSF in Action - Hello, world! - Mozilla
File Edit View Go Bookmarks Tools Window Help
Redisplay
Goodbye

Welcome to JavaServer Faces!

Validation Error: Value is required.

Enter number of controls to display:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
60	61	62	63																

Figure 1.9 The Hello, world! application after you submit a blank value for the required text box field and click the Redisplay button. Because a validation error occurred, the value of the associated JavaBean property didn't change.

you enter the number "99999" into the text box and click the Redisplay button, as shown in figure 1.10.

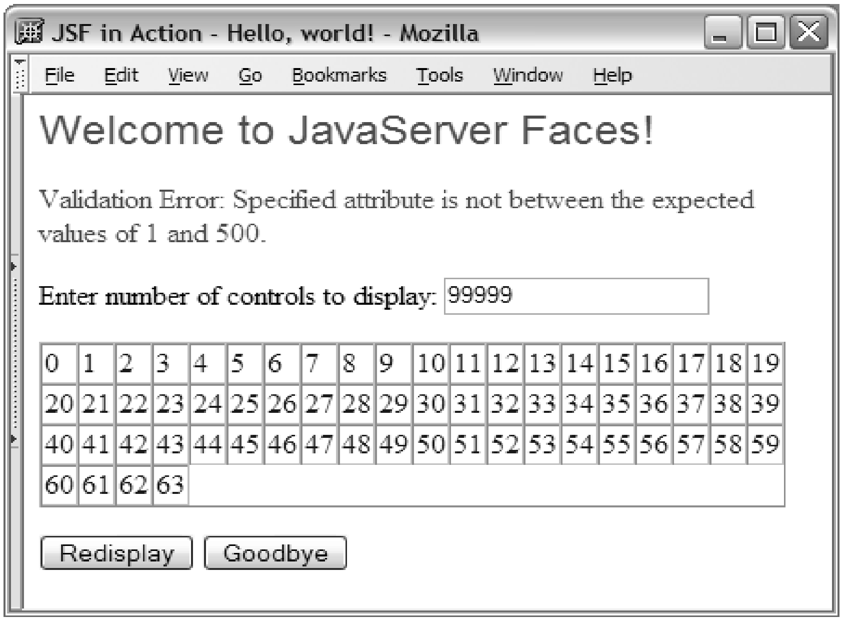
Don't worry about the text of the error messages—in your own applications you can customize it. The important point is that in both cases, when the form was submitted the associated JavaBean property wasn't modified.

If you click the Goodbye button, you see the goodbye.jsp page, shown in figure 1.11. Even though this is an entirely different page, the value of the JavaBean property is displayed. JSF components can reference a JavaBean living in any application scope.

Our Hello, world! example is a standard Java web application, as specified by the Servlet API (it does require the standard Faces libraries, though). All five of these figures were generated with two JSPs. Let's look at them in detail.

1.5.1 Dissecting hello.jsp

Our main page, hello.jsp, provides the interface for figures 1.7 to 1.10. JSF is integrated with JSP through the use of custom tag libraries. The JSF custom tags enable JSPs to use Faces UI components. Tools will often allow you to design JSF pages by dragging and dropping JSF components from a palette. As a matter of


JSF in Action - Hello, world! - Mozilla
File Edit View Go Bookmarks Tools Window Help
Redisplay
Goodbye

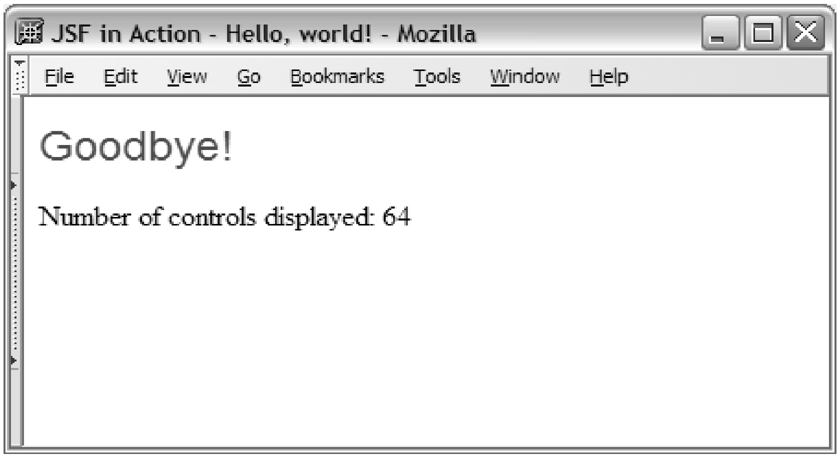
Welcome to JavaServer Faces!

Validation Error: Specified attribute is not between the expected values of 1 and 500.

Enter number of controls to display: 99999

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
60	61	62	63																

Figure 1.10 The Hello, world! application after you enter in the value "99999" into the text box and click the Redisplay button. The field only accepts numbers between 1 and 500, so a validation error is shown. Because a validation error occurred, the value of the associated JavaBean property didn't change.


JSF in Action - Hello, world! - Mozilla
File Edit View Go Bookmarks Tools Window Help

Goodbye!

Number of controls displayed: 64

Figure 1.11 The Hello, world! application after you click the Goodbye button. Note that the JavaBean property, which was synchronized with the text box of the first page, is displayed on this page.

fact, figures 1.1 to 1.3 are screen shots of designing hello.jsp in different IDEs. These IDEs ultimately generate something like listing 1.1 (and, of course, you can create JSF pages by hand).

Listing 1.1 hello.jsp: opening page of our Hello, world! application (browser output shown in figures 1.7–1.10)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>         ❶ JSF tag
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>           libraries

<f:view>          ❷ Tag enclosing
  <html>            all JSF tags
    <head>
      <title>
        JSF in Action - Hello, world!
      </title>
    </head>
    <body>
      <h:form id="welcomeForm">          ❸ HtmlForm
        <h:outputText id="welcomeOutput"     component
                  value="Welcome to JavaServer Faces!"     HtmlOutputText
                  style="font-family: Arial, sans-serif; font-size: 24;   component ❹
                  color: green;"/>
        <p>                                                  HtmlMessage
          <h:message id="errors" for="helloInput" style="color: red"/>   component ❺
        </p>
        <p>
          <h:outputLabel for="helloInput">               HtmlOutputLabel with
            <h:outputText id="helloInputLabel"           child HtmlOutputText ❻
                       value="Enter number of controls to display:"/>
          </h:outputLabel>
          <h:inputText id="helloInput" value="#{helloBean.numControls}"
                    required="true">
            <f:validateLongRange minimum="1" maximum="500"/>   HtmlInputText ❼
          </h:inputText>                                         component
        </p>
        <p>
          <h:panelGrid id="controlPanel"                  HtmlPanelGrid
                    binding="#{helloBean.controlPanel}"     component ❽
                    columns="20" border="1" cellspacing="0"/>
        </p>
        <h:commandButton id="redisplayCommand" type="submit"    HtmlCommandButton
                          value="Redisplay"                        components
                         actionListener="#{helloBean.addControls}"/>   ❾

        <h:commandButton id="goodbyeCommand" type="submit" value="Goodbye"
                         action="#{helloBean.goodbye}" immediate="true"/>
      </h:form>
      </body>
    </html>
  </f:view>
```

❶ First, we import the core JavaServer Faces tag library. This library provides custom tags for such basic tasks as validation and event handling. Next, we import the basic HTML tag library, which provides custom tags for UI components like text boxes, output labels, and forms. (The prefixes "f" and "h" are suggested, but not required.)

❷ The `<f:view>` custom tag must enclose all other Faces-related tags (from both the core tag library and the basic HTML tag library).

❸ The `<h:form>` tag represents an `HtmlForm` component, which is a container for other components and is used for posting information back to the server. You can have more than one `HtmlForm` on the same page, but all input controls must be nested within a `<h:form>` tag.

❹ The `<h:outputText>` tag creates an `HtmlOutputText` component, which simply displays read-only data to the screen. This tag has an `id` attribute as well as a `value` attribute. The `id` attribute is optional for all components; it's not required unless you need to reference the component somewhere else. (Components can be referenced with client-side technologies like JavaScript or in Java code.) The `value` attribute specifies the text you want to display.

❺ The `<h:message>` tag is for the `HtmlMessage` component, which displays validation and conversion errors for a specific component. The `for` attribute tells it to display errors for the control with the identifier `helloInput`, which is the identifier for the text box on the page (❼). If no errors have occurred, nothing is displayed.

❻ The `<h:outputLabel>` tag creates a new `HtmlOutputLabel` component, which is used as a label for input controls. The `for` property associates the label with an input control, which in this case is `helloInput` (❼). `HtmlOutputLabels` don't display anything, so we also need a child `HtmlOutputText` (created by the nested `<h:outputText>` tag) to display the label's text.

❼ The `<h:inputText>` tag is used to create an `HtmlInputText` component that accepts text input. Note that the `value` property is `"#{helloBean.numControls}"`, which is a JSF Expression Language (EL) expression referencing the `numControls` property of a backing bean, called `helloBean`. (The JSF EL is a based upon the EL introduced with JSP 2.0.)

Faces will automatically search the different scopes of the web application (request, session, application) for the specified backing bean. In this case, it will find a bean stored under the key `helloBean` in the application's session. The value of the component and `helloBean`'s `numControls` property are synchronized so that if one changes, the other one will change as well (unless the text in the `HtmlInputText` component is invalid).

Input controls have a `required` property, which determines whether or not the field must have a value. In this case, `required` is set to `true`, so the component will only accept non-empty input. If the user enters an empty value, the page will be redisplayed, and the `HtmlMessage` (❺) component will display an error message, as shown in figure 1.9.

JSF also supports validators, which are responsible for making sure that the user enters an acceptable value. Each input control can be associated with one or more validators. The `<f:validateLongRange>` tag registers a `LongRange` validator for this `HtmlInputText` component. The validator checks to make sure that any input is a number between 1 and 500, inclusive. If the user enters a value outside that range, the validator will reject the input, and the page will be redisplayed with the `HtmlMessage` (❺) component displaying the error message shown in figure 1.10.

Whenever the user's input is rejected, the object referenced by the `HtmlInputText` component's `value` property will not be updated.

❽ An `HtmlPanelGrid` component is represented by the `<h:panelGrid>` tag. `HtmlPanelGrid` represents a configurable container for other components that is displayed as an HTML table.

Any JSF component can be associated directly with a backing bean via its JSP tag's `binding` attribute. (Some tools will do this automatically for all of the components on a page.) The tag's `binding` attribute is set to `"#{helloBean.controlPanel}"`. This is a JSF EL expression that references `helloBean`'s `controlPanel` property, which is of type `HtmlPanelGrid`. This ensures that `helloBean` always has access to the `HtmlPanelGrid` component on the page.

❾ The `<h:commandButton>` specifies an `HtmlCommandButton` component that's displayed as an HTML form button. `HtmlCommandButtons` send action events to the application when they are clicked by a user. The event listener (a method that executes in response to an event) can be directly referenced via the `actionListener` property. The first `HtmlCommandButton`'s `actionListener` property is set to `"#{helloBean.addControls}"`, which is an expression that tells JSF to find the `helloBean` object and then call its `addControls` method to handle the event. Once the method has been executed, the page will be redisplayed.

The second `HtmlCommandButton` has an `action` property set instead of an `actionListener` property. The value of this property, `"#{helloBean.goodbye}"`, references a specialized event listener that handles navigation. This is why clicking on this button loads the goodbye.jsp page instead of redisplaying the hello.jsp page. This button also has the `immediate` property set to `true`, which tells JSF to execute the associated listener before any validations or updates occur. This way, clicking this button still works if the value of the input control is incorrect.

That's it for hello.jsp. Listing 1.2 shows the HTML output after a validation error has occurred (the browser's view is shown in figure 1.10).

Listing 1.2 The HTML output of hello.jsp (this code is the source for figure 1.10)

```
 <html>
    <head>
      <title>
        JSF in Action - Hello, world!
      </title>
    </head>
    <body>
      <form id="welcomeForm" method="post"
            action="/jia-hello-world/faces/hello.jsp"
            enctype="application/x-www-form-urlencoded">

        <span id="welcomeForm:welcomeOutput"
              style="font-family: Arial, sans-serif; font-size: 24
                     color: green;">Welcome to
JavaServer Faces!</span>
        <p>
         <span id="welcomeForm:errors" style="color: red">
Validation Error: Specified attribute is not between the expected values
 of 1 and 500.</span>
        </p>
        <p>
          <label for="welcomeForm:helloInput">
            <span id="welcomeForm:helloInputLabel">
Enter number of controls to display:</span>
          </label>
          <input id="welcomeForm:helloInput" type="text"
                 name="welcomeForm:helloInput" value="99999"/>
        </p>
        <p>
          <table id="welcomeForm:controlPanel" border="1" cellspacing="0">
            <tbody>
              <tr>
                <td><span style="color: blue"> 0 </span></td>
             ...
                <td><span style="color: blue"> 19 </span></td>
              </tr>
             <tr>
               <td><span style="color: blue"> 20 </span></td>
                 ...
               <td><span style="color: blue"> 39 </span></td>
             </tr>
```

(1) **HtmlForm component**

HtmlOutputText component

HtmlMessage component (2)

HtmlOutputLabel with HtmlOutputText

HtmlInputText component

(3) **HtmlPanelGrid component**

```
            <tr>
              <td><span style="color: blue"> 40 </span></td>
              ...
              <td><span style="color: blue"> 59 </span></td>
           </tr>
            <tr>
              <td><span style="color: blue"> 60 </span></td>
              ...
              <td><span style="color: blue"> 63 </span></td>
            </tr>
           </tbody>
         </table>
       </p>
       <input id="welcomeForm:redisplayCommand" type="submit"
              name="welcomeForm:redisplayCommand" value="Redisplay" />

       <input id="welcomeForm:goodbyeCommand" type="submit"
              name="welcomeForm:goodbyeCommand" value="Goodbye" />
       ...
     </form>
   </body>
 </html>
```

❸ **HtmlPanelGrid component**

HtmlCommandButton components

You can see from the listing that every component defined in the JSP has a representation in the displayed HTML page. Note that the `<h:form>` tag (❶), which represents an `HtmlForm` component, has an `action` attribute that actually points back to the calling JSP but with the preface "faces". This is an alias for the Faces servlet, which is defined in the application's deployment descriptor. Redisplaying the calling page is the default behavior, but a Faces application can also navigate to another page (which is what happens when the user clicks the Goodbye button).

The output of the `HtmlMessage` component (❷) is the text "Validation Error: Specified attribute is not between the expected values of 1 and 500." As you might expect, this message was generated by the `LongRange` validator we registered in the JSP page. When the validator rejected the attempt to post an incorrect value, the validator created a new error message and the framework refrained from updating the associated JavaBean property's value.

Each HTML element that maps to a JSF component has an `id` attribute that's derived from the `id` specified in the JSP (if no `id` is specified, one will be created automatically). This is called the *client identifier*, and it's what JSF uses to map an input value to a component on the server. Some components also use the `name` attribute for the client identifier.

The output of the `HtmlPanelGrid` component (❸) is an HTML table. Note that the `border` and `cellspacing` properties specified in the JSP were passed through directly to the HTML. (Most of the standard HTML components expose HTML-specific properties that are simply passed through to the browser.) Each cell in the table is the output of an `HtmlOutputText` component that was added to the `HtmlPanelGrid` in Java code, in response to a user clicking the Redisplay button. (In the real HTML, there are 64 cells because that's the number that was entered into the text box; we left some of them out of the listing because, well, that's a lot of lot of extra paper!)

We'll examine the Java code soon enough, but let's look at goodbye.jsp first.

1.5.2 Dissecting goodbye.jsp

The goodbye.jsp page, shown in figure 1.11, is displayed when the user clicks the Goodbye button. The page (listing 1.3) contains some of the same elements as the hello.jsp page: imports for the JSF tag libraries, an `HtmlForm` component, and `HtmlOutputText` components. One of the `HtmlOutputText` components references the same `helloBean` object as the previous page. This works fine because the object lives in the application's session and consequently survives between page requests.

Listing 1.3 goodbye.jsp: Closing page of our Hello, world! application (the browser output is shown in figure 1.11)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>

<f:view>
  <html>
    <head>
      <title>
        JSF in Action - Hello, world!
      </title>
    </head>
    <body>
      <h:form id="goodbyeForm">
        <p>
          <h:outputText id="welcomeOutput" value="Goodbye!"
                    style="font-family: Arial, sans-serif; font-size: 24;
                    font-style: bold; color: green;"/>
        </p>
        <p>
          <h:outputText id="helloBeanOutputLabel"
                      value="Number of controls displayed:"/>
          <h:outputText id="helloBeanOutput"                    | Same backing
                      value="#{helloBean.numControls}"/>        | bean as hello.jsp
        </p>
      </h:form>
    </body>
  </html>
</f:view>
```

There's nothing special about the HTML generated by this page that we didn't cover in the previous section, so we'll spare you the details. What's important is that we were able to build a functional application with validation and page navigation with only two simple JSPs. (If we didn't want to show navigation, the first page would have been good enough.)

Now, let's look at the code behind these pages.

1.5.3 Examining the HelloBean class

Both hello.jsp and goodbye.jsp contain JSF components that reference a backing bean called `helloBean` through JSF EL expressions. This single JavaBean contains everything needed for this application: two properties and two methods. It's shown in listing 1.4.

Listing 1.4 HelloBean.java: The simple backing bean for our Hello, world! application

```
package org.jia.hello;

import javax.faces.application.Application;
import javax.faces.component.html.HtmlOutputText;
import javax.faces.component.html.HtmlPanelGrid;
import javax.faces.context.FacesContext;
import javax.faces.event.ActionEvent;

import java.util.List;

public class HelloBean   <-1 No required superclass
{
  private int numControls;
  private HtmlPanelGrid controlPanel;

  public int getNumControls()
  {
    return numControls;
  }                                            2 Property
                                                 referenced on
  public void setNumControls(int numControls)    both JSPs
  {
    this.numControls = numControls;
  }
```

```
    public HtmlPanelGrid getControlPanel()
    {
      return controlPanel;
    }

    public void setControlPanel(HtmlPanelGrid controlPanel)
    {
      this.controlPanel = controlPanel;
    }

    public void addControls(ActionEvent actionEvent)
    {
      Application application =
          FacesContext.getCurrentInstance().getApplication();
      List children = controlPanel.getChildren();
      children.clear();
      for (int count = 0; count < numControls; count++)
      {
        HtmlOutputText output = (HtmlOutputText)application.
                          createComponent(HtmlOutputText.COMPONENT_TYPE);
        output.setValue(" " + count + " ");
        output.setStyle("color: blue");
        children.add(output);
      }
    }

    public String goodbye()
    {
      return "success";
    }
  }
```

❸ Property bound to HtmlPanelGrid

❹ Executed by Redisplay HtmlCommandButton

❺ Executed by Goodbye HtmlCommandButton

❶ Unlike a lot of other frameworks, JSF backing beans don't have to inherit from a specific class. They simply need to expose their properties using ordinary Java-Bean conventions and use specific signatures for their event-handling methods.

❷ The `numControls` property is referenced by the `HtmlInputText` component on hello.jsp and an `HtmlOutputText` component on goodbye.jsp. Whenever the user changes the value in the `HtmlInputText` component, the value of this property is changed as well (if the input is valid).

❸ The `controlPanel` property is of type `HtmlPanelGrid`, which is the actual Java class created by the `<h:panelGrid>` tag used in hello.jsp. That tag's `binding` attribute associates the component instance created by the tag with the `control-Panel` property. This allows `HelloBean` to manipulate the actual code—a task it happily performs in ❹.

❹ `addControls` is a method designed to handle action events (an *action listener method*); you can tell, because it accepts an `ActionEvent` as its only parameter. The Redisplay `HtmlCommandButton` on hello.jsp references this method with its `actionListener` property. This tells JSF to execute the method when handling the action event generated when the user clicks the Redisplay button. (Associating a component with an event listener *method* may seem strange if you're used to frameworks like Swing that always require a separate event listener *interface*. JSF supports interface-style listeners as well, but the preferred method is to use listener methods because they alleviate the need for adapter classes in backing beans.)

When this method is executed, it adds a new `HtmlOutputText` component to the `controlPanel` `numControls` times (clearing it first). So, if the value of `numControls` is 64, as it is in our example, this code will create and add 64 `HtmlOutputText` instances to `controlPanel`. Each instance's value is set to equal its number in the sequence, starting at zero and ending at 64. And finally, each instance's `style` property is set to `"color: blue"`.

Because `controlPanel` is an `HtmlPanelGrid` instance, it will display all of these child controls inside an HTML table; each `HtmlOutputText` component wll be displayed in a single cell of the table. Figure 1.8 shows what `controlPanel` looks like after this method has executed.

❺ Like `addControls`, the `goodbye` method is a type of event listener. However, it is associated with JSF's navigation system, so its job is to return a string, or a logical outcome, that the navigation system can use to determine which page to load next. These types of methods are called *action methods*.

The `goodbye` method is associated with the Goodbye `HtmlCommandButton` on hello.jsp via its `action` property. So when a user clicks the Goodbye button, the `goodbye` method is executed. In this case, `goodbye` doesn't do any work to determine the logical outcome; it just returns `"success"`. This outcome is associated with a specific page in a Faces configuration file, which we cover next.

Because `goodbye` doesn't perform any processing (as it would in a real application), we could have achieved the same effect by hardcoding the text `"success"` in the button's `action` property. This is because the navigation system will either use the literal value of an `HtmlCommandButton`'s `action` property or the outcome of an action method (if the property references one).

1.5.4 Configuration with faces-config.xml

Like most frameworks, Faces has a configuration file; it's called, believe it or not, faces-config.xml. (Technically, JSF supports multiple configuration files, but we'll keep things simple for now.) This XML file allows you to define rules for navigation, initialize JavaBeans, register your own custom JSF components and validators, and configure several other aspects of a JSF application. This simple application requires configuration only for bean initialization and navigation; the file is shown in listing 1.5.

Listing 1.5 faces-config.xml: The Faces configuration file for Hello, world!

```
<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">

<faces-config>    <-❶ Encloses all configuration elements

  <managed-bean>
    <description>The one and only HelloBean.</description>
    <managed-bean-name>helloBean</managed-bean-name>
    <managed-bean-class>org.jia.hello.HelloBean
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>

  <navigation-rule>
    <description>Navigation from the hello page.</description>
    <from-view-id>/hello.jsp</from-view-id>
    <navigation-case>
      <from-outcome>success</from-outcome>
      <to-view-id>/goodbye.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>

</faces-config>
```

❷ **Declares HelloBean in the session**

❸ **Declares navigation case**

First and foremost, a JSF configuration file is an XML document whose root node is `<faces-config>` (❶). In this file, you can declare one or more JavaBeans for use in your application. You can give each one a name (which can be referenced via JSF EL expressions), a description, and a scope, and you can even initialize its properties. Objects declared in a configuration file are called *managed beans*. In the listing, we have declared the `helloBean` object used throughout the Hello, world! application (❷). Note that the name of the object is “helloBean”, which is the same name used in JSF EL expressions on the two JSPs. The class is `org.jia.hello.HelloBean`, which is the name of the backing bean class we examined in the previous section. The managed bean name and the object’s class name don’t have to be the same.

Declaring navigation is as simple as declaring a managed bean. Each JSF application can have one or more navigation rules. A *navigation rule* specifies the possible routes from a given page. Each route is called a *navigation case*. The listing shows the navigation rule for Hello, world!'s hello.jsp page (❸). hello.jsp has a Goodbye button that loads another page, so there is a single navigation case: if the outcome is `"success"`, the page goodbye.jsp will be displayed. This outcome is returned from `helloBean`'s `goodbye` method, which is executed when a user clicks the Goodbye button.

It's worthwhile to point out that some aspects of JSF configuration, particularly navigation, can be handled visually with tools. Now, let's see how our application is configured at the web application level.

1.5.5 Configuration with web.xml

All J2EE web applications are configured with a web.xml deployment descriptor; Faces applications are no different. However, JSF applications require that you specify the `FacesServlet`, which is usually the main servlet for the application. In addition, requests must be mapped to this servlet. The deployment descriptor for our Hello, world! application is shown in listing 1.6. You can expect some tools to generate the required JSF-related elements for you.

Listing 1.6 web.xml: The deployment descriptor for our Hello, world! application

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3/
   /EN" "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <display-name>Hello, World!</display-name>
  <description>Welcome to JavaServer Faces</description>

  <servlet>                                                              | JSF
    <servlet-name>Faces Servlet</servlet-name>                           | servlet
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>       |
    <load-on-startup>1</load-on-startup>                                 |
  </servlet>                                                             |
  <servlet-mapping>                                  | Standard JSF
    <servlet-name>Faces Servlet</servlet-name>       | mapping
    <url-pattern>/faces/*</url-pattern>              |
  </servlet-mapping>                                 |
</web-app>
```

That's it—Hello, world! dissected. You can see that JSF does a lot of things for you—validation, event handling, navigation, UI component management, and so on. As we walk through the various aspects of JSF in more detail, you'll gain a deep understanding of all of the services it provides so that you can concentrate on building the application and avoid that joyous thing they call grunt work.

1.6 Summary

JavaServer Faces (JSF, or "Faces") is a UI framework for building Java web applications; it was developed through the Java Community Process (JCP) and will become part of Java 2 Enterprise Edition (J2EE). One of the main goals of Faces is to bring the RAD style of application development, made popular by tools like Microsoft Visual Basic and Borland Delphi, to the world of Java web applications.

JSF provides a set of standard widgets (buttons, hyperlinks, checkboxes, and so on), a model for creating custom widgets, a way to process client-generated events on the server, and excellent tool support. You can even synchronize a UI component with an object's value, which eliminates a lot of tedious code.

All JSF applications are built on top of the Servlet API, communicate via HTTP, and use a display technology like JSP. JavaServer Faces applications don't *require* JSP, though. They can use technologies like XML/XSLT, other template engines, or plain Java code. However, Faces implementations are required to provide basic integration with JSP, so most of the examples in this book are in JSP.

The component architecture of Faces leverages JavaBeans for properties, fundamental tool support, an event model, and several other goodies. JSF is considered a web application framework because it performs a lot of common development tasks so that developers can focus on more fun things like business logic. One of the key features is support of the Model 2 design pattern, which enforces separation of presentation and business logic code. However, Faces focuses on UI components and events. As such, it integrates quite nicely with the other frameworks like Struts, and overlaps quite a bit with the functionality of higher-level frameworks.

The Hello, world! example demonstrates the basic aspects of a JavaServer Faces application. It shows how easy it is to define a UI with components like text boxes, labels, and buttons. It also shows how Faces automatically handles input validation and updating a JavaBean based on the value of a text control.

In the next chapter, we'll look at the core JSF concepts and examine how the framework masks the request/response nature of HTTP.

JAVA

JavaServer Faces IN ACTION

Kito Mann

JavaServer Faces is the new big thing in Java web development. It improves your power and reduces your workload through the use of UI components and events, instead of HTTP requests and responses. JSF components—buttons, text boxes, checkboxes, data grids, etc.—live between user requests, which eliminates the hassle of maintaining state. JSF also synchronizes user input with application objects, automating another tedious aspect of web development.

JavaServer Faces in Action is an introduction, a tutorial, and a handy reference. With the help of many examples, the book explains what JSF is, how it works, and how it relates to other frameworks and technologies like Struts, Servlets, Portlets, JSP, and JSTL. It provides detailed coverage of standard components, renderers, converters, and validators, and how to use them to create solid applications. This book will help you start building JSF solutions today.

What's Inside

- A gentle introduction
- JSF under the hood
- Using JSF widgets
- How to:
  - integrate with Struts and existing apps
  - benefit from JSF tools from Oracle, IBM, and Sun
  - build custom components (lots of examples)
  - build renderers, converters, validators
  - put it all together in a JSF application

A developer for 16 years, **Kito D. Mann** is an enterprise architect who has consulted for several Fortune 500 companies. He runs the JSFCentral.com community site. Kito lives in Stamford, Connecticut with his wife, two parrots, and four cats.

"I can't wait to make it available to the people I teach."

—Sang Shin, Java Technology Evangelist
Sun Microsystems Inc.

"This book unlocks the full power of JSF... It's a necessity."

—Jonas Jacobi
Senior Product Manager, Oracle

"Gets right into using JSF and explains the advanced topics in detail. Well-written and a quick read."

—Matthew Schmidt
Director, Advanced Technology
Javalobby

"A book written by a programmer who knows what programmers need."

—Alex Kolundzija
Front-End Developer
Columbia House

"... an excellent job showing that JSF can be used with other technologies. A great reference and tutorial!"

—Mike Nash, JSF Expert Group Member
Author, *Explorer's Guide to Java Open Source Tools*

AUTHOR ONLINE Ask the Author

Ebook edition

www.manning.com/mann

54295
9 781932 394122
ISBN 1-932394-12-5

MANNING $44.95 US/$62.95 Canada